U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

2°ano – MIEIC – Abril 2017

Recolha de Lixo Inteligente

Conceção e Análise de Algoritmos

Turma 6 - Grupo C

Diogo Peixoto Pereira – up201504326
Gonçalo Vasconcelos Cunha Miranda Moreno – up201503871
Maria Eduarda Santos Cunha – up201506524

Índice

1. Introdução

O sistema atual de recolha de lixo consiste, tipicamente, na passagem de um camião de resíduos por todos os pontos de recolha espalhados numa cidade, recolhendo o conteúdo dos contentores e transportando-o para uma estação de tratamento de resíduos, que lida com os vários tipos adequadamente. Existem contentores específicos para cada tipo de resíduo, geralmente, amarelo para o plástico e embalagens, azul para papel e cartão, verde para vidro, vermelho para pilhas e, por fim, o preto para resíduos domésticos indiferenciados.

O nosso objetivo com este trabalho é a simulação de um sistema de recolha mais avançado, que, ao invés de obrigar um camião a esta recolha de forma não prática e ineficaz, leve a um percurso muito mais eficiente.

Propõe-se, então, para começar que a nossa implementação seja efetuada assumindo a existência de contentores equipados com sensores de volume. Estes sensores permitem saber quando é que o contentor precisa efetivamente de ser coletado, permitindo um trajeto de viagem que tenha em conta a quantidade de resíduos presente em cada contentor.

Consideramos ainda dois tipos de frotas, uma homogénea e outra heterogénea. A primeira, com um só tipo de veículo de recolha e, a segunda, com veículos para cada tipo de resíduo.

2. Formalização do Problema

A problematização a ser resolvida é a criação de trajetórias eficientes em termos de maximizar o número de pontos de recolha que cada camião percorre quando sai da garagem, dada também a sua própria capacidade, e ter em conta qual o camião e centro de tratamento mais adequados dada essa trajetória e o tipo de resíduos a recolher.

Essencialmente, o nosso problema encaixa-se no *Vehicle Routing Problem*, mais especificamente no *Capacitated Vehicle Routing Problem*.

Dada a complexidade de resolução deste problema, optamos por dividi-lo em duas partes:

1. Uma fase de pré-processamento, em que se escolhem os contentores com base no seu volume de resíduos, de modo a que não se ultrapasse a capacidade do camião. Subdividindo em várias viagens até que todos sejam recolhidos.
2. Calcular a melhor rota desde a garagem e, de seguida, a partir do aterro sanitário.

2.1. Dados de Entrada

Um mapa, representado por um grafo, G = (V, E), cujos nós são aleatoriamente determinados como garagens, contentores ou aterros sanitários.

G – Grafo que abstrai mapa.

V – Vértices/nós que simbolizam as garagens, contentores e aterros sanitários.

$E_{i,j}$ – Arestas que ligam os vários vértices, V_i a V_j, com *labels* que indicam a distância entre eles.

O mapa utilizado como exemplo provem do *Open Street Maps* e é representativo de uma zona de Espinho, que escolhemos dada a sua organização em ruas perpendiculares, o que facilita, em muito, a confirmação do bom funcionamento do nosso código.

Este mapa foi traduzido em três ficheiros de texto através do software *OSM2TXT Parser*, sendo que o primeiro contem informação acerca dos nós, o segundo sobre as arestas e o terceiro referente às ligações entre nós.

2.2. Função Objetivo

Como o nosso problema não se trata do *Travelling Salesman Problem* clássico, por começar num nó e acabar noutro e também por não termos as distâncias diretas entre todos os nós, é extremamente improvável encontrar a solução ótima. No entanto, esperamos que ao minimizar a equação deste algoritmo, a do *Vehicle Routing Problem* seja minimizada como consequência. Logo, a função que realisticamente estamos a minimizar é a seguinte:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}$$

2.3. Restrições

Como restrição, apresenta-se a capacidade do camião.

A nível de restrições matemáticas, existem as restrições específicas do *TSP*:

1. $0 \leq x_{ij} \leq 1$ i, j = 1, ..., n;
2. $u_i \in \mathrm{Z}$ i = 1, ..., n;
3. $\sum_{i=1, i \neq j}^{n} x_{ij} = 1$ j = 1, ..., n;

4. $\sum_{j=1,j\neq i}^{n} x_{ij} = 1$ i = 1, ..., n;

5. $u_i - u_j + nx_{ij} \leq n - 1$ $2 \leq i \neq j \leq n$.

2.4. Resultados Esperados

Uma rota que, começando na garagem, percorra todos os contentores e acabe no aterro sanitário.

3. Solução

Como já referimos na formalização do problema, a solução pode ser dividida em duas fases.

A fase de pré-processamento consiste em:

1. Definir a garagem e o aterro;
2. Usar a abordagem *Depth-first* para eliminar nós inacessíveis;
3. Definir contentores do lixo;
4. Através dos algoritmos de *Dijsktra* ou *Floyd-Marshall*, calcular a distância de todos os contentores a todos os outros contentores. Aproximando, assim, o nosso problema do *Travelling Salesman Problem*;

```
1  function dijkstraShortestPath(T source):
2
3  map<T, double> dis
4  map<T, T> prev
5
6  create Priority Queue, Q
7
8  for each vertex in Graph:
9   if info ≠ source then
10   dis ← (info, INT_INFINITY)
11   prev ← (info, -1)
12  else
13   dis[source] ← 0
14  end if
15 end for
16
17 Q.add_with_priority(info, dis[info])
18
19 while Q is not empty:
20  u_top ← Q.extractTop()
21  Vertex<T>* u ← this->getVertex(u_top)
22
23  for each neighbour v of u:
24   Vertex<T> * v ← u->adj.at(i).dest
25   alt ← dis[u->info] + u->adj.at(i).weight
26
27   if (alt < dis[v->info]) then
28    dis[v->info] ← alt
29    prev[v->info] ← u->info
30    Q.decreasePriority(v->info, alt)
31   end if
32  end for
```

```
33 end while
34
35 for each vertex:
36  v->path ← getVertex(prev[v->info])
37 end for
```

A fase seguinte, correspondente ao cálculo do ***Travelling Salesman Problem***:

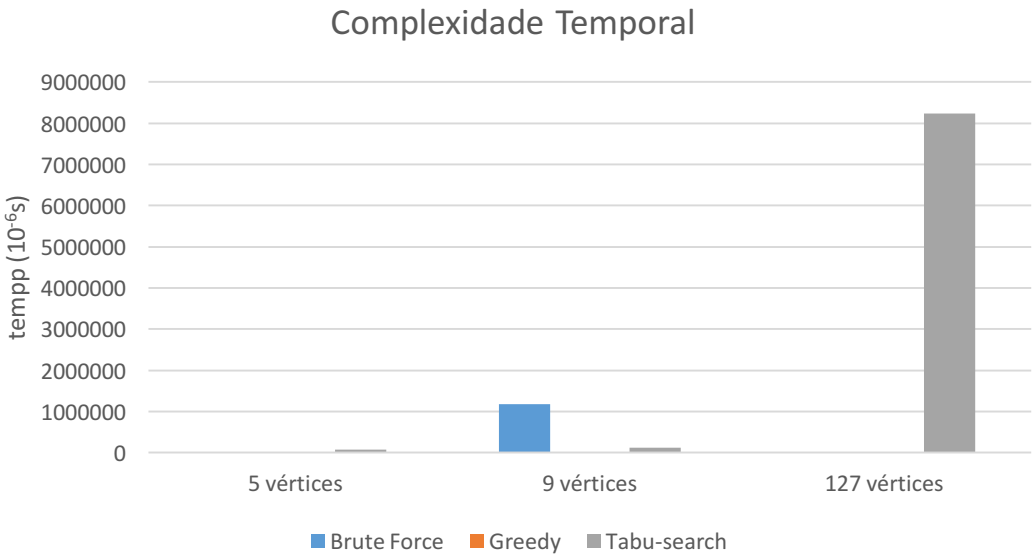
1. Escolher contentores possíveis, numa viagem, cujo conteúdo não ultrapasse a capacidade do camião de recolha, pelo algoritmo *Brute-force*, *Greedy* ou *Tabu-search*.

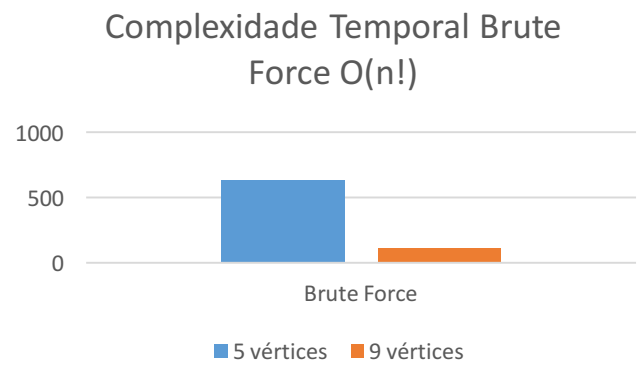
Nota: A função de *Tabu-search* implementada é semelhante às funções do mesmo tipo para o *TSP*, mas com a particularidade de, em vez de, a cada iteração, as jogadas tabus passarem a válidas, são necessárias duas iterações para o mesmo efeito.

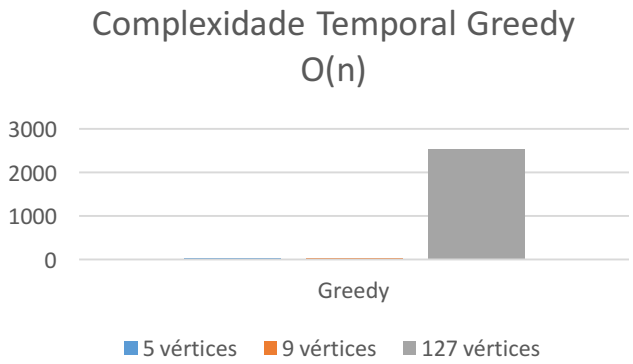
2. Guardar resultado num vetor correspondente a uma viagem;

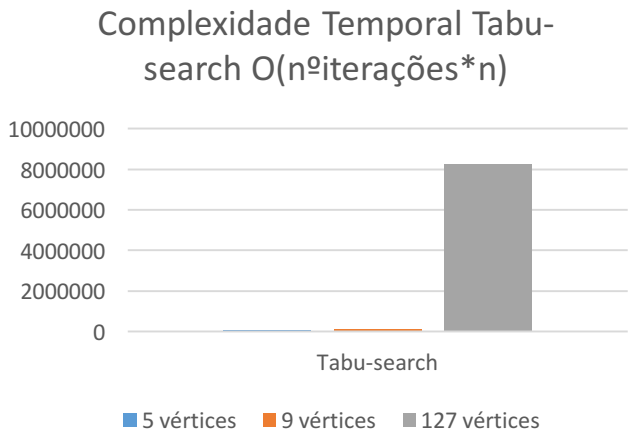
3. Repetir o processo até todos os contentores terem sido esvaziados;

4. Reconstruir rota tendo em conta o grafo inicial proveniente do *Open Street Map*.









- Não calculamos *TSP* por *Brute Force* com 127 vértices, dado que a complexidade de O(n!) iria resultar num número superior à idade do Universo (13,8 biliões de anos).
- Os grafos de 127 nós foram feitos através de um mapa de uma zona de Espinho do *Open Street Map*. Para o algoritmo *Tabu*-search recorremos a 1000 iterações.
- Os algoritmos de resolução do *TSP* não ocupam espaço mais do próprio grafo, a não ser o *Tabu-search*, que cresce com o número de iterações.

4. Diagrama de Classes

5. Casos de Utilização

1. Dado o foro genérico do programa, resolvendo apenas de forma aproximada o *Vehicle Routing Problem*, poderia ser utilizado para otimizar vários tipos de rotas, como, por exemplo, a do enunciado (recolha de lixo), entrega de mercadorias...

A aplicação deste programa traria aumento de eficiência ao reduzir o tempo de viagem e os gastos em combustível;

2. Visualização de um mapa com o *GraphViewer* e respetiva tradução em ficheiros de texto com informação relativa a nós e arestas.

6. Dificuldades

A primeira dificuldade a ter em conta será termos percebido em que categoria geral o nosso problema se encontrava. Foi bastante intuitivo perceber que se enquadrava no *Vehicle Routing Problem*, mas não tão fácil assumir que teríamos de recorrer também ao *Travelling Salesman Problem*. Isto, pois este último envolve o conhecimento da distância de um nó a todos os outros, para todos os nós, que é informação que, à partida, não possuíamos, já que o nosso grafo inicial só nos dava indicação das distâncias entre cada dois nós, a partir das *labels*. Após percebermos a necessidade desta informação, desenvolvemos o segundo grafo com nós selecionados a partir da informação que calculamos.

Ao longo do projeto, adotamos a prática dos testes unitários, mais direcionada para os algoritmos, o que nos custou algum tempo, mas cremos que terá compensado em termos globais, pois permitia a alteração e correção de código constante tendo consciência dos erros consequentes dessas mudanças no código. Assim, eliminamos a necessidade de *debugging* a cada erro que surgisse em código já implementado e previamente funcional.

Ao tentar implementar o algoritmo de *Branch and Bound*, não conseguíamos encontrar um *lower bound* para uma dada rota, pois, para esse efeito, é necessário calcular a *minimum spanning tree*.

Ainda, tentamos escrever o algoritmo que calcula a distância de todos os contentores usando *Disjktra*, com *threads*, mas não conseguimos que fosse funcional dado que não chegamos a implementar o construtor de cópia para a classe Grafo.

7. Contributos

Diogo Pereira - 33%

Gonçalo Moreno - 33%

Eduarda Cunha - 33%

8. Conclusão

O tipo de problema como o que nos foi proposto é extremamente complicado em termos de computação e os algoritmos disponíveis não conseguem satisfazer por completo a sua resolução.

Ainda que o mais rápido que encontramos tenha sido o *Greedy*, a solução oferecida por este não é muito boa. Poderíamos tentar melhorá-la com um algoritmo genético (ou semelhante) do tipo *Tabu-search* ou *Swarm Optimization* (que não chegamos a ter oportunidade de implementar), mas esta busca é difícil para mais de cem nós.

9. Bibliografia

[1] Narahari, Y. "8.4.2 Optimal Solution for TSP using Branch and Bound". Game Theory Lab. http://lcm.csa.iisc.ernet.in/dsa/node187.html (last accessed April 6, 2017)

[2] Stack Overflow. http://stackoverflow.com/questions/22985590/calculating-the-held-karp-lower-bound-for-the-traveling-salesmantsp

[3] MIT. "Branch and Bound". MIT OPEN COURSE WARE. https://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/tutorials/MIT15_053S13_tut10.pdf (last accessed April 4, 2017)

[4] "Travelling Salesman problem". Wikipedia. https://en.wikipedia.org/wiki/Travelling_salesman_problem#Related_problems

[5] Stack Overflow. http://stackoverflow.com/questions/22985590/calculating-the-held-karp-lower-bound-for-the-traveling-salesmantsp

[6] Kumar Singhal, Ritesh; Pandey, Dr. D. K. "Approximation of Shortest Path using Travelling Salesman Problem". International Journal of Advanced and Innovative Research. http://ijair.jctjournals.com/oct2012/t121015.pdf

[7] "Dijkstra's algorithm". Wikipedia. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithmAGENTS

[8] Poole, David; Mackworth, Alan. "3.7.4 Branch and Bound". Artificial Intelligence. http://artint.info/html/ArtInt_63.html

[9] Gao, Jiyao. "Branch and bound (BB)". Northwestern Univeristy Process Optimization Open Textbook. https://optimization.mccormick.northwestern.edu/index.php/Branch_and_bound_(BB)